Validating SystemC Implementations Against Their Formal Specifications

Jannis Stoppe[2] Robert Wille[1,2] Rolf Drechsler[1,2]

[1]Institute of Computer Science, University of Bremen, 28359 Bremen, Germany
[2]Cyber-Physical Systems, DFKI GmbH, 28359 Bremen, Germany
{jstoppe,rwille,drechsler}@uni-bremen.de

ABSTRACT

The ever increasing complexity of embedded systems leads to a constant strive for higher levels of abstraction. While the design at the *Electronic System Level (ESL)* with SystemC as the common programming language is state-of-the-art today, also the use of formal specifications by means of modeling languages such as UML or SysML receives more and more attention. This raises the question of how to validate an ESL implementation against a given formal specification. For this, SystemC's limited introspection and reflection features pose a serious obstacle. In this paper, a methodology is presented that retrieves the necessary static and dynamic information which is needed in order to validate a SystemC design. For this purpose, we retrieve information from the SystemC API and compiler-generated debug symbols. The proposed solution can be applied to a wide variety of project setups and requires only minimal adjustments to retrieve the necessary information.

1. INTRODUCTION

With the increasing complexity of embedded systems, corresponding design methods are constantly faced with new challenges. In order to catch up with this development and to tackle the ever increasing complexity, higher levels of abstraction are introduced to the design process. The current state-of-the-art assumes design at the ESL [15] with its defacto standard language SystemC [14, 17, 19] as the typical starting point of the implementation process. However, in accordance with the strive for higher levels of abstractions, also the use of formal specifications prior to the implementation receives more and more attention – leading to the design at the *Formal Specification Level* (FSL) [6].

Here, modeling languages such as the *Unified Modeling Language* (UML) [18] or the *Systems Modeling Language* (SysML) [21] are applied to formally specify the structure and the behavior of a system to be implemented. These FSL descriptions work as precise blueprint for the designer and can even be exploited to generate ESL code stubs or initial implementations. However, manual refinement of the resulting ESL implementation is usually required afterwards. Due to this manual process, new errors might be introduced. Consequently, the ESL implementation might violate constraints that are formulated in the formal specification.

In this work, we propose an approach which validates an implemented ESL description against its formal specification. For this purpose, a methodology is suggested which

- *extracts system states* ("snapshots") before and during the execution of a SystemC design and then
- *compares* those states with the previously specified FSL description.

The severely limited capabilities for introspection and reflection of SystemC represent a significant obstacle for this goal. In fact, extracting the respective system states is crucial to the proposed methodology. We address this problem by utilizing the existing SystemC API (allowing for retrieving dynamic information) and the compiler-generated debug symbols (providing access to static information). This does not assume any changes and restrictions on the compiler. Hence, the resulting approach can be applied to a wide variety of project setups, requiring only minimal adjustments. A case study illustrates how this, in fact, helps to detect errors and inconsistencies of an ESL implementation compared to its FSL specification.

The remainder of this paper is structured as follows: The next section provides a brief review on formal specifications at the FSL and SystemC implementations at the ESL. It also introduces the design of an arbiter which works as running example in the rest of this paper. Section 3 motivates the considered problem and introduces the general idea of the proposed methodology. Afterwards, the implementation of it is described in Section 4. Finally, the paper concludes with an evaluation and a sketch of application in Section 5 as well as conclusions in Section 6.

2. PRELIMINARIES

In order to keep this work self-contained, the basics on formal specifications at the FSL and SystemC at the ESL are briefly reviewed in this section. Besides that, the running example is introduced.

2.1 Formal Specifications

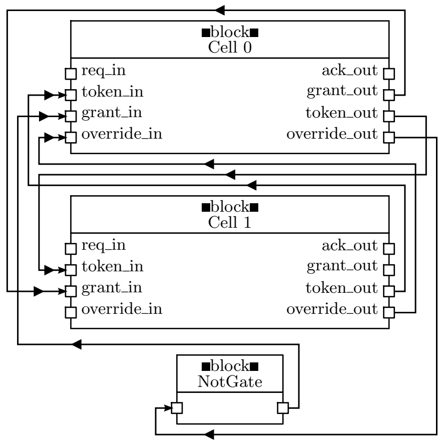
Modeling languages such as the *Unified Modeling Language* (UML) [18] or the *Systems Modeling Language* (SysML) [21] have been established to explicitly specify the design of systems prior to the implementation phase. In this work, we focus on block definition diagrams and sequence diagrams. *Block definition diagrams* formally describe the structure of a system. They are composed of blocks which are organized by compartments to store different information such as attributes, operations, or relations to other blocks. Attributes describe the data elements of the block, whereas operations are used to modify them. The dynamic flow caused by operation calls can be visualized by *sequence diagrams*. Here, instances of the blocks are extended by life lines that express the time of creation and destruction in the scenario. Arrows indicate operations that are called on an instance and are drawn from the caller to the callee.

Examples of both description means are provided in Fig. 1 – representing the specification of an arbiter which is used as running example in the rest of this paper[1]. The arbiter is a collection of cells that handle the access to an underlying resource (e.g. a bus). These cells are connected to their neighbors and form a circle that has a token traveling around. The cell that currently holds that token grants access to the element connected to it. More precisely, Fig. 1a shows the structure of this system. It contains a two-cell arbiter which would result in a switching behavior, i.e. alternating access between `Cell 0` and `Cell 1`. Fig. 1b illustrates a simple case of a sequence diagram: Assuming that all cells keep requesting the token in a two-cell arbiter, it should be granted alternatingly. `Cell 0` opens the resource to the requesting element and grants the resource to the next cell in the next step. Afterwards, `Cell 1` grants the resource for the first time and, afterwards, returns the token back to `Cell 0`. After that, the process continues analogously.

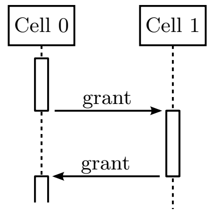
2.2 SystemC

For system design at the ESL, SystemC has been established as a de-facto standard [14, 17, 19]. SystemC is implemented as a C++ library that provides structures to implement and simulate embedded systems early in the design process – e.g. prior to a hardware/software partitioning. For this purpose, SystemC provides additional descriptions means for hardware elements (e.g. modules, signals, etc.) which are instantiated during an *elaboration phase*. In this phase, the execution of a SystemC program is started and the simulated system is instantiated. With the start of the simulation, the *simulation phase* commences, i.e. the instantiated system is executed with the given parameters. This phase lasts until the program terminates. No new modules may be instantiated during this phase.

[1]The specification of this arbiter is inspired by the implementation originally proposed in [5].



(a) Block definition diagram.



(b) Sequence diagram.

Figure 1: Formal specification of the arbiter [5].

Due to the fact of SystemC being a C++ library, all C++ description means may be used during the elaboration phase to instantiate the desired system. This means that a given SystemC program does not necessarily contain the description of only a single system to be simulated. The result of the elaboration phase may depend on user inputs or other external stimuli. The precise nature of the respectively simulated system is, therefore, only known at run-time. The arbiter program e.g. contains a variable that specifies the number of cells which could be read from user input just as well.

3. VALIDATING SYSTEMC AGAINST A FORMAL SPECIFICATION

This section motivates the need for solutions validating SystemC implementations against their formal specifications and sketches the general idea of the solution proposed in this work. The problem is formulated first followed by an overview of the related work on introspection and reflection of SystemC designs. These works build a cornerstone in the proposed validation methodology which is sketched in the end of this section.

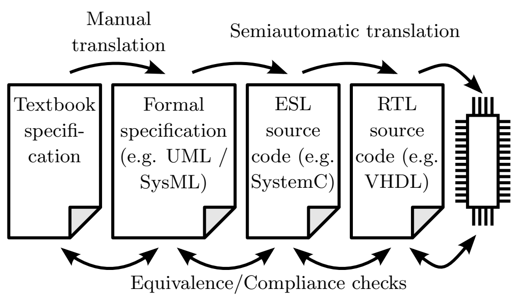

Manual translation
Semiautomatic translation
Textbook specification
Formal specification (e.g. UML / SysML)
ESL source code (e.g. SystemC)
RTL source code (e.g. VHDL)
Equivalence/Compliance checks

Figure 2: Today's design flow.

3.1 Problem Formulation

In order to close the gap from the given specification (provided in natural language) to the initial implementation (at the ESL) and in accordance with the strive for higher levels of abstractions, the use of formal specifications prior to the initial implementations receives more and more attention. Modeling languages such as the UML [21] or the SysML [18] are applied for this purpose. This led to the design at the *Formal Specification Level* (FSL, see e.g. [6]).

FSL specifications offer a formal description of the structure and the behavior of the system to be implemented. Due to this formal representation, code generation techniques can be applied to automatically create ESL stubs and even initial implementations (as indicated by the upper arrows in Fig. 2). But since each of these initial implementations are manually extended afterwards, new errors might be introduced. As a consequence, the resulting (final) ESL implementation might differ from the FSL specification in both, structure and behavior. Hence, after completing the ESL implementation, it is useful to validate the result against the original FSL description (as indicated by the lower arrows in Fig. 2).

Such a process is quite common in the design of software systems. As an example, in [8] a Java implementation is analyzed and extracted back to an UML model. Then, the resulting UML model can be used in order to check whether the software system has been implemented as intended. However, considering the design of embedded systems, a serious obstacle occurs: Implementations in SystemC – the de-facto standard in ESL design – hardly provide any support for introspection and reflection. This significantly hinders the extraction of corresponding models or comparisons to given formal specifications. Motivated by this, we consider the question *"How to validate whether an ESL implementation in SystemC indeed adheres to a corresponding FSL specification?"*. For this purpose, recently developed methods for introspection and reflection of SystemC are utilized and extended.

3.2 Introspection and Reflection of SystemC

Usually, the structure and the behavior of an implementation can be extracted using the language's introspection and reflection features. The former allows to examine the type and/or properties of objects at run time, the latter enables the designer to modify those values and/or invoke methods of the retrieved objects, also at run time. Since SystemC is implemented as a C++ library, it is limited to the native features of C++. Unlike other languages that provide a native support for introspection and reflection (like C# with Esys.net [3, 12, 13] or Java with HJJ [9]), C++ (and, thus, SystemC) are limited to very restricted means concerning the reflection of an implementation's structure and state. Although SystemC does remedy this a bit by providing an API to retrieve instances of SystemC objects at run-time, this is hardly sufficient to compare an implementation to an FSL model. In fact, additional information concerning their static properties (e.g. contents of their functions, a list of members etc.) is needed for a more complete model of the system.

To retrieve these static information, custom SystemC parsers have been proposed in the past (see e.g. [1,2,4,7,11]). They are specialized in the analysis of the SystemC source code in order to retrieve as much information about the design as possible. However, as any C++ construct can be used to create the instances that represent the simulated hardware, these parsers face the severe limitations that (1) some code elements are not trivial to analyze statically (macros, recursions, loops) and (2) some information may not be known at compile-time at all (contents of external files, dynamically linked libraries, command-line parameters). Because of this, a complete design extraction at runtime based on these methods is infeasible.

As an alternative, an approach relying on a customization of the compiler and, by this, enabling static data-extraction at compile-time has been presented in [16]. Of course, providing a custom compiler also can be used to modify the resulting binary to track and report any run-time behaviour and, by this, extract the dynamic information like object instances or variable assignments. However, this results in the consideration of a restricted sub-set of C++; project setups that utilize other compilers or even other versions need to be modified in order to work on the given setup.

Hence, in this work we utilize an idea presented in [20] which exploits the compiler-generated debug symbols in order to retrieve the desired information. During compilation, debug symbols are generated to provide debuggers with static information about the code while the developer debugs the program. As the compiler needs to store this information until the program is executed, this data can also be accessed during run-time. By this, dynamic information on the instantiation of the system (e.g. "there are ten instances of the module `Cell`") and their corresponding static information (e.g. "each module `Cell` has a method `void ctrl_fsm()`") can both be obtained. The usage of the dynamic information makes this approach a simulation-based approach, as the off-the-shelf SystemC kernel is used to generate the instances during the elaboration phase and the following simulation is used to extract consecutive snapshots. How the retrieved data can be utilized is sketched next.

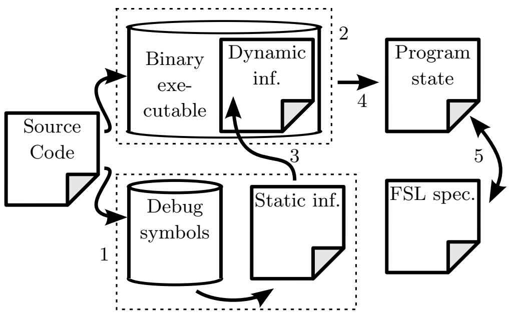


Figure 3: Proposed methodology.

3.3 Proposed Methodology

In order to validate an ESL implementation against a formal specification, the structure and the behavior of the ESL design has to be extracted first. Due to the shortcomings of introspection and reflection in SystemC (as discussed above), the idea of using debug symbols proposed in [20] is utilized for this purpose. More precisely, an information extraction flow as sketched in Fig. 3 is applied.

First, static information on the SystemC implementation is determined exploiting the compiler-generated debug symbols (1). Afterwards, the original code is compiled so that the resulting binary can be executed (2). Both steps do not assume any changes and restrictions on the compiler. As discussed above, SystemC allows then to retrieve information of the instantiated instances at run-time. By additionally exploiting the static information, which is now available through the debug symbols (3), the structure and current state of the executed system can be derived (4). This does not only include which classes/modules have been instantiated, but also the members these classes are composed of (including referenced objects) and which values they have been assigned with. During the execution of the SystemC implementation, several such "snapshots" of the system states can be conducted. From those, corresponding FSL descriptions (e.g. a block definition diagram of the structure and a sequence diagram including the respective object diagrams of the behavior) can be derived and eventually compared to the originally given FSL specification (5). For this purpose, existing modeling frameworks such as e.g. the *Eclipse Modeling Framework* can be utilized as they offer corresponding model checking capabilities out of the box. For this work, a simple checker that asserts all parts of the specification are present in the implementation was implemented as a proof-of-concept.

4. IMPLEMENTATION

This section describes the implementation of the proposed methodology in detail. As sketched above and in Fig. 3, validating a SystemC implementation against a given FSL specification is conducted in five steps:

1. retrieve static information,
2. retrieve module instantiations ,
3. link static and dynamic information,
4. derive the current program state, and
5. compare the program states to the FSL specification.

All these steps are detailed in the following. To this end, we use the FSL specification and the SystemC implementation of the arbiter introduced in Section 2 as a running example.

4.1 Retrieve Static Information

Retrieving the static structure of the ESL implementation is crucial for a structural analysis of the instantiations during the execution of a SystemC program. More precisely, their static layout, i.e. their classes, members, etc., is needed. As discussed in Section 3.2, this information is stored in the debug symbols which are generated during the compile process. Accordingly, the static information can be retrieved by simply parsing these debug symbols. As, to our knowledge, there is no compiler that does not save its debug symbols for later use, this methodology should be applicable universally, i.e. without any restrictions on supported language constructs or compiler versions. Moreover, usually proper parsers or an API are provided for an easy access. While the results of this step may change for each compilation (if the code was changed), it remains the same if the resulting binary is executed repeatedly. This step is therefore executed after building the SystemC project.

4.2 Retrieve Module Instantiations

The static information retrieved in Step 1 does not unveil how often certain classes are instantiated. While it specifies that there is a module of type `cell`, the amount of modules used in the design as well as the connections between them and other modules is not part of SystemC's static features. This information is only available during run-time. It therefore needs to be extracted dynamically.

For this purpose, SystemC offers an API to retrieve a list of all modules, signals etc. that have been instantiated during SystemC's elaboration phase. More precisely, an `sc_object_manager`-instance can be accessed, providing a list of all module instances. These do not only form the core of the simulated system but can also be used as an initial nucleus for an (incomplete) block definition diagram. Utilizing the `sc_object_manager`-instance for the SystemC implementation of the considered arbiter leads to the incomplete block definition diagram as shown in Fig. 4. As can be seen, the module instances and connections between them can be extracted automatically using this method. Similarities with the originally given FSL specification from Fig. 1a are already evident. However, details e.g. on the members and their current assignment are still missing.

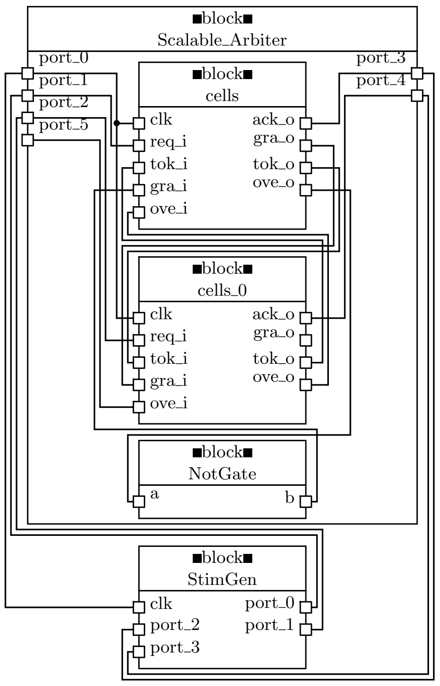


Figure 4: Retrieved module instantiations.

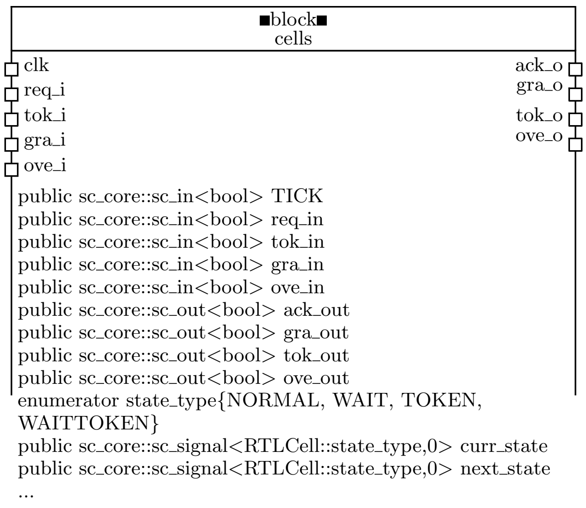
4.3 Link Static and Dynamic Information

The objects retrieved in Step 2 now need to be connected to the static information of their respective classes retrieved in Step 1. The SystemC API returns `sc_object` instances. While the extraction of information based on the `sc_object` type might be helpful already, extracting the information contained in the fields that are members of the actually instantiated class (that inherits `sc_object` in some way) is preferable. For this, however, the actual type needs to be retrieved from the instance that resides in the memory during the execution.

To access them, *Run-Time Type Inspection* (RTTI) is applied which is part of the current C++ standard [10] and, hence, the closest solution for a universally applicable type retrieval system. The derived (dynamic) type names can be matched to the names contained in the debug symbols that got extracted in Step 1. Once the static and dynamic types have been linked, the resulting data consists of SystemC object instances that do contain information about their static fields (i.e. methods, variables etc.) but not the currently assigned values. Applied to the running example, this allows to retrieve the respective members of the single components as exemplarily been sketched for one object of type `Cell` in Fig. 5.

4.4 Derive the Current Program State

In order to complete the "snapshot" of a currently considered program state, the respective assignments to each member have to be derived. This is accomplished by traversing the memory allocated by the considered SystemC execution.



```
public sc_core::sc_in<bool> TICK
public sc_core::sc_in<bool> req_in
public sc_core::sc_in<bool> tok_in
public sc_core::sc_in<bool> gra_in
public sc_core::sc_in<bool> ove_in
public sc_core::sc_out<bool> ack_out
public sc_core::sc_out<bool> gra_out
public sc_core::sc_out<bool> tok_out
public sc_core::sc_out<bool> ove_out
enumerator state_type{NORMAL, WAIT, TOKEN,
WAITTOKEN}
public sc_core::sc_signal<RTLCell::state_type,0> curr_state
public sc_core::sc_signal<RTLCell::state_type,0> next_state
...
```

Figure 5: Retrieved arbiter cell w/ static inform.

A breadth-first-search along a running program's references is applied, starting with the objects that were retrieved in Step 2. The information contained in the debug symbols is used to determine an individual object's memory layout. All fields are located and separated into base types (like integer, float, etc.) to extract their values and compound types and pointers, both of which are again enqueued for later analysis.

However, this traversal of allocated memory does have the following issues:

4.4.1 Bad Pointers

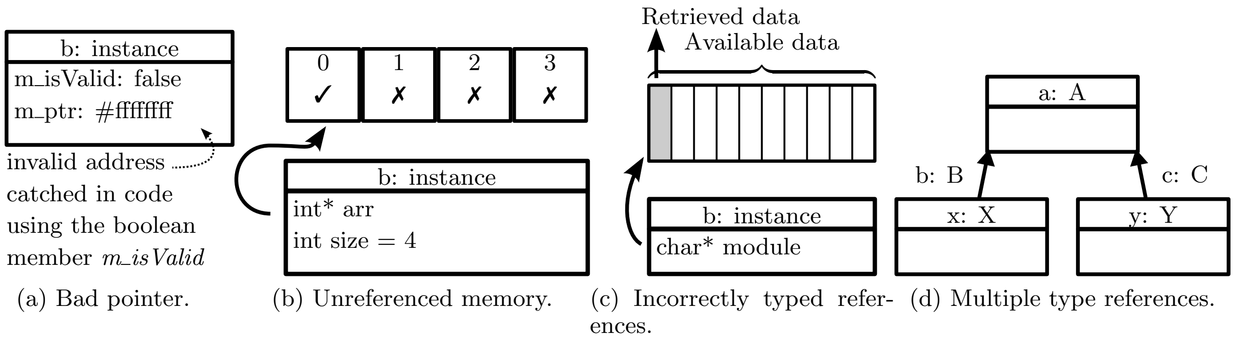
Null pointers are not analyzed further. A more serious problem are bad pointers. For example, Fig. 6a shows an instance with a bad pointer. Although a field indicates that this pointer should not be used, this semantic distinction cannot be recognized automatically. Hence, our implementation assumes that all pointers are valid and catches memory access violations at run-time. Due to the read-only nature of the data extraction, no data stored in the memory is compromised by this.

4.4.2 Unreferenced Memory

There might be parts in memory that are allocated and used but not addressed. A common example is an array that is created on the heap and addressed using a pointer while a second value stores its size. As an example, see Fig. 6b showing an instance with an array of size 4. In this case, only the referred memory is considered, i.e. we traverse the graph that is formed by elements in memory and their connections only. Elements that are separated from this graph cannot be reached by the traversal algorithm and, hence, are ignored. That is, in the example of Fig. 6b `arr[0]` is extracted as a single int value (as both, pointer and type, are known), but `arr[1]` to `arr[3]` are ignored.

4.4.3 Incorrectly Typed References

Objects referenced using an erroneous type also pose a problem for the proposed extraction method. Our approach



(a) Bad pointer. (b) Unreferenced memory. (c) Incorrectly typed references. (d) Multiple type references.

Figure 6: Special issues during program state extraction.

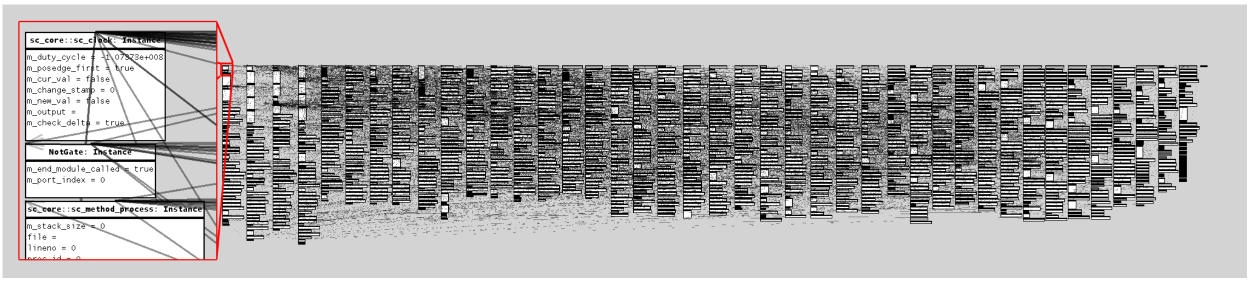


Figure 7: Resulting program state. The area on the left is a close-up to the upper-left corner of the diagram. While a manual comparison would be hard due to the amount of data that is retrieved, this data can be used as a foundation for an automated model-checking approach.

assumes that the type of a pointer corresponds to the type of data that is actually stored at the given address. Due to C++'s lack of type safety, an instance of type a may be stored in a part of the memory that is referenced as being of a certain unrelated type b. Our approach will then assume that the address contains an instance of this type b and, therefore, extracts the data as if it would be of that type b. This may result in missing or incorrectly extracted data. For example, Fig. 6c shows a pointer of type `char` that actually points to an instance that contains much more data. However, our approach treats this as a `char` and, hence only retrieves the first byte.

4.4.4 Multiple Type References

Although similar to the incorrectly typed references, this is more a problem concerning established modeling languages: If a single address in memory is addressed by two or more differently typed variables (as illustrated in Fig. 6d), a corresponding representation is usually not available in established modeling languages. Currently, our approach just exports all interpretations of a certain variable. Depending on the desired modeling language, these features might need additional tweaking.

The result of this traversal is indicated in Fig. 7 for the running example, i.e. the arbiter implementation[2]. The resulting model does not only contain the instances of the original specification shown in Fig. 1a, but also all instances of any object that is referenced through these (directly or indirectly) and the values that are currently assigned to the according member variables. Although this is much more data than actually needed, it now provides a precise "snapshot" of the current system state in terms of an FSL description. By sequentially applying this scheme for each system state which shall be compared against an FSL specification, all the information needed for this comparison is retrieved.

[2]Please note that this figure is not supposed to provide detailed information but serves as an illustration of the magnitude of the obtained information.

4.5 Compare the Program States to the FSL Specification

The extracted models shall now be compared to a given FSL specification. As the static and dynamic information that were extracted from the memory can simply be translated e.g. to block definition diagrams (as already illustrated above), checking for compliance with a given model is simple. Existing modeling frameworks such as the *Eclipse Modeling Framework* can be utilized for this purpose.

Validating the behavior instead requires some manual additions. As the proposed method only extracts program states ("snapshots") for certain points in the execution of the program and is unable to supervise a certain behavior, checking if a certain protocol is adhered cannot be performed fully automatically. In fact, the designer has to explicitly enforce when a snapshot shall be retrieved. If e.g. the behavior is specified by means of a sequence diagram, then a snapshot after each operation call (which changes the state of the system) is appropriate. By this, the respective states of the implementation are retrieved and can be validated against the specification. This of course neither guarantees that the change in state is a result of the communication nor that the given communication was indeed the only behavior occurring between those states, but valid checks in these places indicate whether an implementation complies with its formal specification.

5. EVALUATION & APPLICATION

The proposed methodology has been implemented and evaluated. In this section, we discuss the performance of the resulting approach and illustrate how its application advances the design of embedded systems.

5.1 Performance

The suggested method is able to extract a significant amount of information to be used for the validation of the structure and the behavior of an ESL implementation against a corresponding FSL specification. The performance depends thereby on what exactly is extracted. The determination of the static information of the considered arbiter required e.g. approx. one minute on an AMD Phenom II X4 965 with 8GB RAM – most of the time is thereby spent on writing the information onto the disc (with output disabled, the process takes approx. 12 seconds). Retrieving a snapshot, e.g. the one from Fig. 7, required approx. 14 CPU seconds. Note that, in the current proof-of-concept implementation, information is stored in an external XML file that needs to be parsed and searched during the dynamic extraction. Hence, further improvements in the performance are very likely but were not in the scope of this work. For a variety of SystemC designs, the respective information was successfully retrieved.

Besides that, the proposed solution

- can essentially be transferred to any setup, as long as readable debug symbols and RTTI data are generated by the compiler,
- is able to extract not only the system's modules but also any other C++ objects to which they are connected to,
- relies on the SystemC API to retrieve the objects to be inspected and analyzed, i.e. the user does not have to add any additional code apart from the statement that denotes at which points during the execution a snapshot is required,
- does not add any overhead to the execution until the extraction statements are executed.

Overall, the determined FSL descriptions from a SystemC execution can be used for the validation of running SystemC designs without a manual translation into unit tests or something similar. How this advances the design of embedded systems is exemplarily illustrated next.

5.2 Application

The proposed approach has been applied in order to validate the SystemC implementation from [5] against its formal specification provided in Section 2. For this purpose, a simple model to state matcher was implemented which utilizes the retrieved information and compares them to the originally given FSL specification. This enabled the detection of the following errors and inconsistencies:

- Additional wrapper modules have been added to the SystemC implementation of the respective cells. Furthermore, the original specification does not contain the module `Scalable_Arbiter` that hosts the cells within.
- The identifiers of the blocks and its corresponding modules did not match. That is, identifiers `Cell 0`, `Cell 1`, . . . , `Cell n-1` are used in the specification, while the identifiers `cells`, `cells_0`, `cells_1`, . . . , `cells_n-2` are used in the SystemC counterpart. This is a crucial issue as designers might likely map e.g. `Cell 2` to `cells_2` (though it should be mapped to `cells_1`).
- Similarly, the identifiers of the inputs and outputs did not match. The implementation abbreviated the names, i.e. the identifier `override_out` became an `ove_o`.
- The implementation contains inputs for the clock, several other inputs for the wrapper module, and a `StimGen`-module that was obviously added for testing the setup by generating signal stimuli. All that is not part of the formal specification. In particular, the addition of clock inputs is crucial as this changes the interface of the module.
- A consecutive "snapshot" of the state after each clock cycle showed that the implementation does not make the token of the arbiter travel around the cells. This obviously is a serious design error which needs to be inspected.

After explicitely pin-pointed to it, most of these issues can also be seen by comparing Fig. 1a and Fig. 4. Of course, these issues are not necessarily errors. But they clearly emphasize parts of the design which are different from the original specification. In particular with increasing complexity of the designs, this provides a helpful tool which warns the designer about potential discrepancies and possible sources for confusion later on in the design process.

6. CONCLUSION

In this work, we use the potential resulting from the application of high level formal specifications by providing an approach that allows for the validation of an ESL implementation against its formal specification. For this purpose, a solution has been proposed which overcomes the limited introspection and reflection features of SystemC. This enables the derivation of snapshots during the execution of a SystemC program which can be compared against a given FSL specification. A case study illustrated the usefulness of the presented methodology.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research (BMBF) within the project SPECifIC under grant no. 01IW13001 as well as the German Research Foundation (DFG) within a Reinhart Koselleck project under grant no. DR 287/23-1 and a research project under grant no. WI 3401/5-1.

7. REFERENCES

[1] David Berner, Jean-Pierre Talpin, Hiren Patel, Deepak Abraham Mathaikutty, and Sandeep Shukla. SystemCXML: An extensible SystemC front end using XML. In *Proceedings of the Forum on Specification and Design Languages*, pages 99–104, 2005.

[2] Carlo Brandolese, Paolini Di Felice, Luigi Pomante, and Daniele Scarpazza. Parsing SystemC: An Open-Source, Easy-to-Extend Parser. In *IADIS International Conference on Applied Computing*, pages 706–709, 2006.

[3] Olivier Brassard, Frederic Rousseau, Jean David, Mathieu Kastle, and El Aboulhamid. Automatic Generation of Embedded Systems with .NET Framework Based Tools. *2006 IEEE North-East Workshop on Circuits and Systems*, pages 165–168, June 2006.

[4] Javier Castillo, Pablo Huerta, and Jose Ignacio Martinez. An open-source tool for SystemC to Verilog automatic translation. *Latin American Applied Research*, 37(1):53–58, 2007.

[5] Rolf Drechsler and Daniel Grosse. Reachability Analysis for Formal Verification of SystemC. In *Euromicro Symposium on Digital System Design 2002*, pages 337–340, 2002.

[6] Rolf Drechsler, Mathias Soeken, and Robert Wille. Formal Specification Level: Towards verification-driven design based on natural language processing. In *Forum on Specification and Design Languages*, pages 53–58, 2012.

[7] Görschwin Fey, Daniel Große, Tim Cassens, Christian Genz, Tim Warode, and Rolf Drechsler. ParSyC: an efficient SystemC parser. In *Workshop on Synthesis And System Integration of Mixed Information technologies*, pages 148–154, 2004.

[8] Martin Gogolla and Ralf Kollmann. Re-Documentation of Java with UML Class Diagrams. In *7th Reengineering Forum*, pages 41–48, 2000.

[9] John Hopf, G. Stewart Itzstein, and David Kearney. Hardware Join Java: a high level language for reconfigurable hardware development. In *International Conference on Field-Programmable Technology*, pages 344–347, 2002.

[10] ISO. Information technology – Programming languages – C++. Norm ISO/IEC 14882:2011, International Organization for Standardization, 2000.

[11] FZI Karlsruhe. KaSCPar - Karlsruhe SystemC Parser Suite, 2012. http://www.fzi.de/index.php/de/component/content/article/238
ispe-sim/4350-kascpar-karlsruhe-systemc-parser-suite.

[12] James Lapalme, El Mostapha Aboulhamid, Gabriela Nicolescu, Luc Charest, Francois R. Boyer, Jean Pierre David, and Guy Bois. ESys. Net: a new solution for embedded systems modeling and simulation. *ACM SIGPLAN Notices*, 39(7):107–114, 2004.

[13] James Lapalme, El Mostapha Aboulhamid, Gabriela Nicolescu, Luc Charest, Francois R. Boyer, Jean Pierre David, and Guy Bois. .NET framework - a solution for the next generation tools for system-level modeling and simulation. In *Design, Automation and Test in Europe Conference*, pages 732–733, 2004.

[14] Kevin Marquet, Matthieu Moy, and Bageshri Karkare. A theoretical and experimental review of SystemC front-ends. In *Forum on Specification and Design Languages*, pages 124–129, 2010.

[15] Grant Martin, Brian Bailey, and Andrew Piziali. *ESL Design and Verification: A Prescription for Electronic System Level Methodology.* Morgan Kaufmann, 2007.

[16] Matthieu Moy, Florence Maraninchi, and Laurent Maillet-Contoz. PINAPA: An Extraction Tool for SystemC Descriptions of Systems-on-a-Chip. In *Conference on Embedded software*, pages 317–324, 2005.

[17] O.S.C. Initiative. IEEE Standard SystemC Language Reference Manual. *IEEE Computer Society*, 2006.

[18] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language reference manual.* Addison-Wesley Longman, Essex, UK, January 1999.

[19] Carsten Schulz-Key, Markus Winterholer, Thomas Schweizer, Tommy Kuhn, and Wolfgang Rosenstiel. Object-oriented modeling and synthesis of SystemC specifications. In *Asia and South Pacific Design Automation Conference*, pages 238–243, 2004.

[20] Jannis Stoppe, Robert Wille, and Rolf Drechsler. Data Extraction from SystemC Designs using Debug Symbols and the SystemC API. In *IEEE Computer Society Annual Symposium on VLSI*, pages 26–31, 2013.

[21] Tim Weilkiens. *Systems Engineering with SysML/UML: Modeling, Analysis, Design.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, February 2008.